Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 10: this Friday @ 11:59 pm
  - Lab 8: Tuesday 3/28 @ 11:59 pm
  - Assignment 3: Friday 3/31 @ 11:59 pm
    - Support video and slides on Canvas

Previous lecture

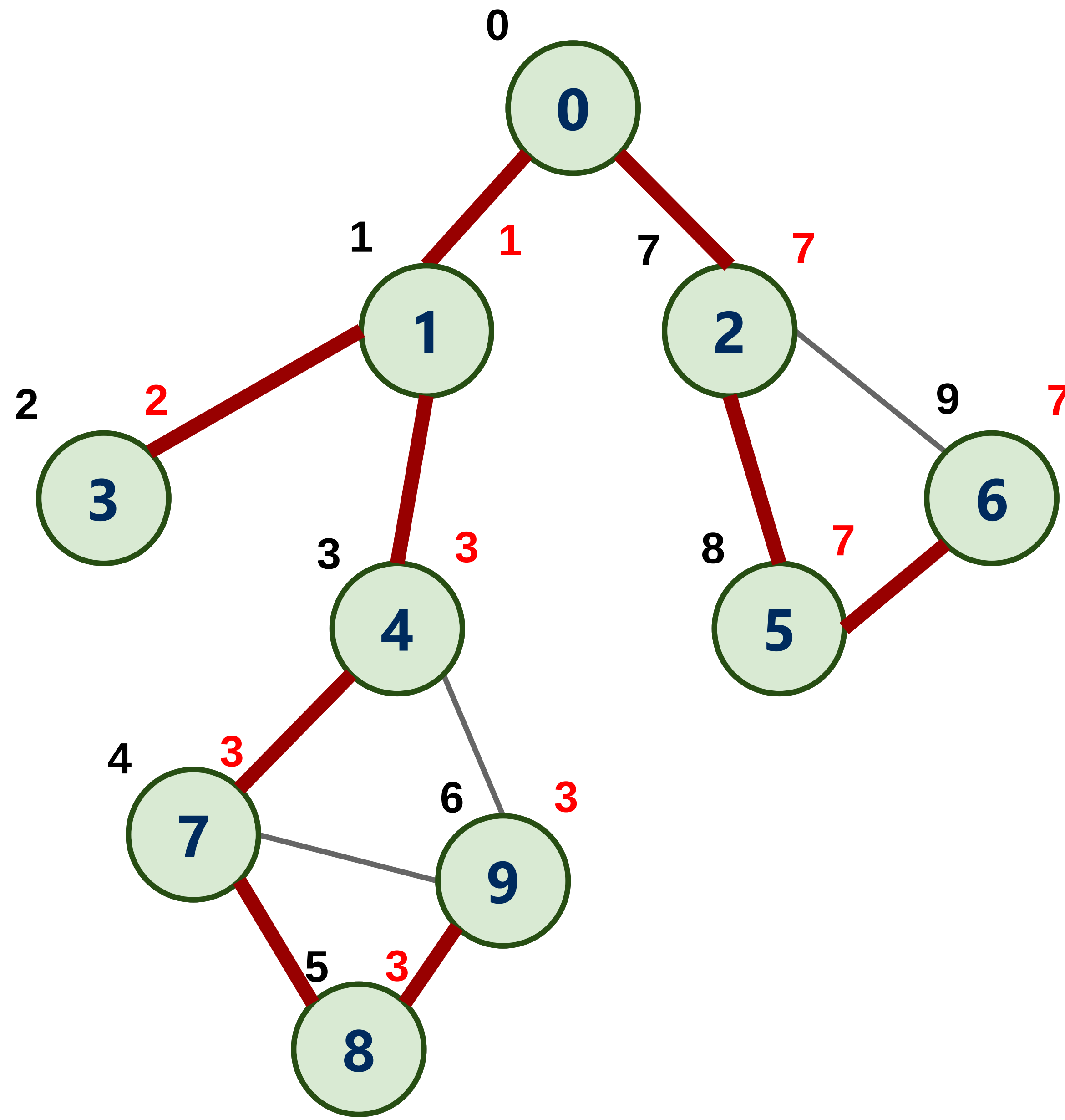
- ADT Graph
  - traversals
    - BFS
      - shortest paths based on number of edges
      - connected components
    - DFS
      - finding articulation points of a graph

This Lecture

- ADT Graph
  - DFS
    - finding articulation points of a graph
- Repetitive Minimum Problem

low(v)

- How do we find low(v)?
- low(v) = Min of:
  - num(v)
  - num(w) for all back edges (v, w)
  - low(w) of all children of *v*

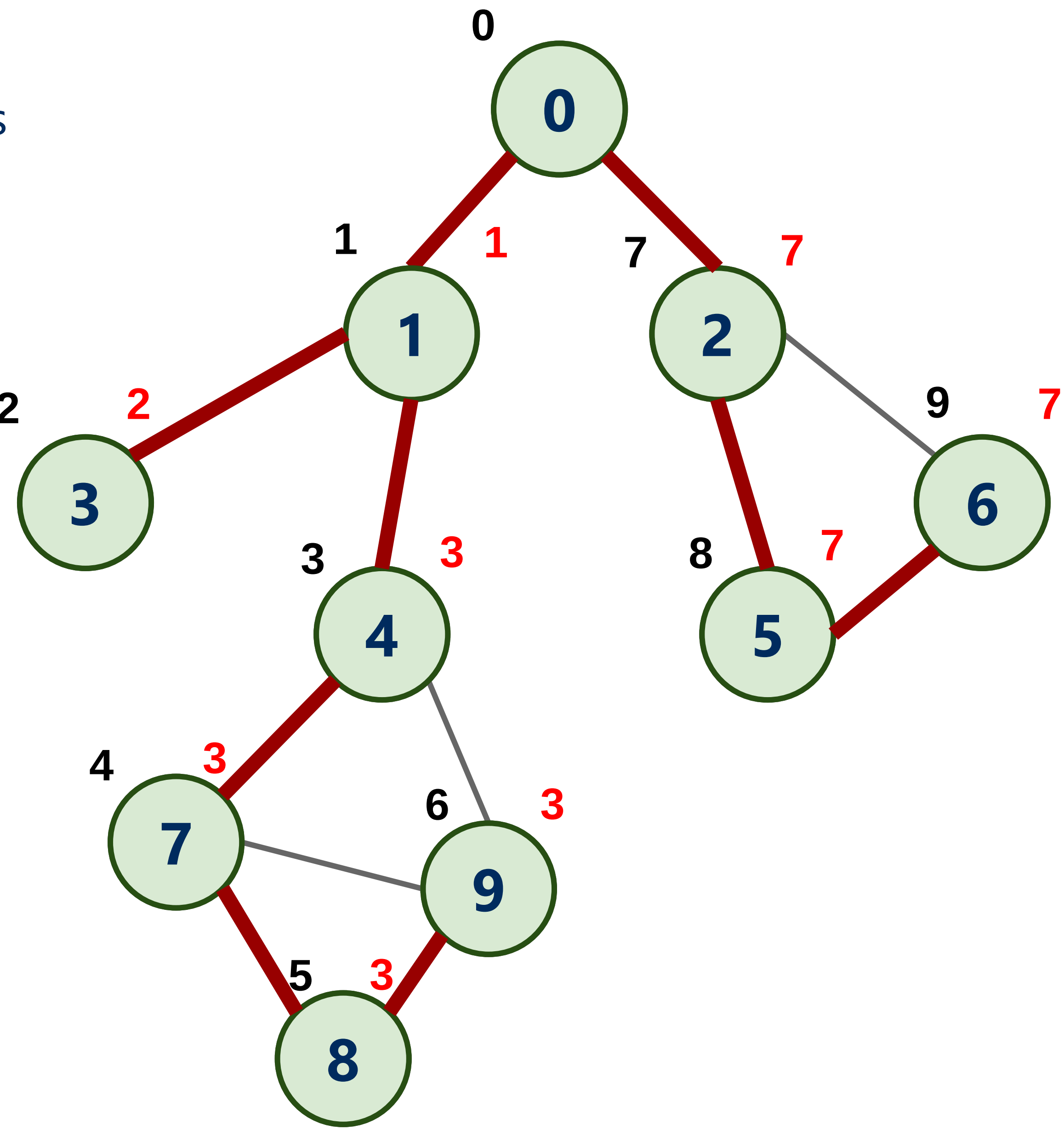


low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

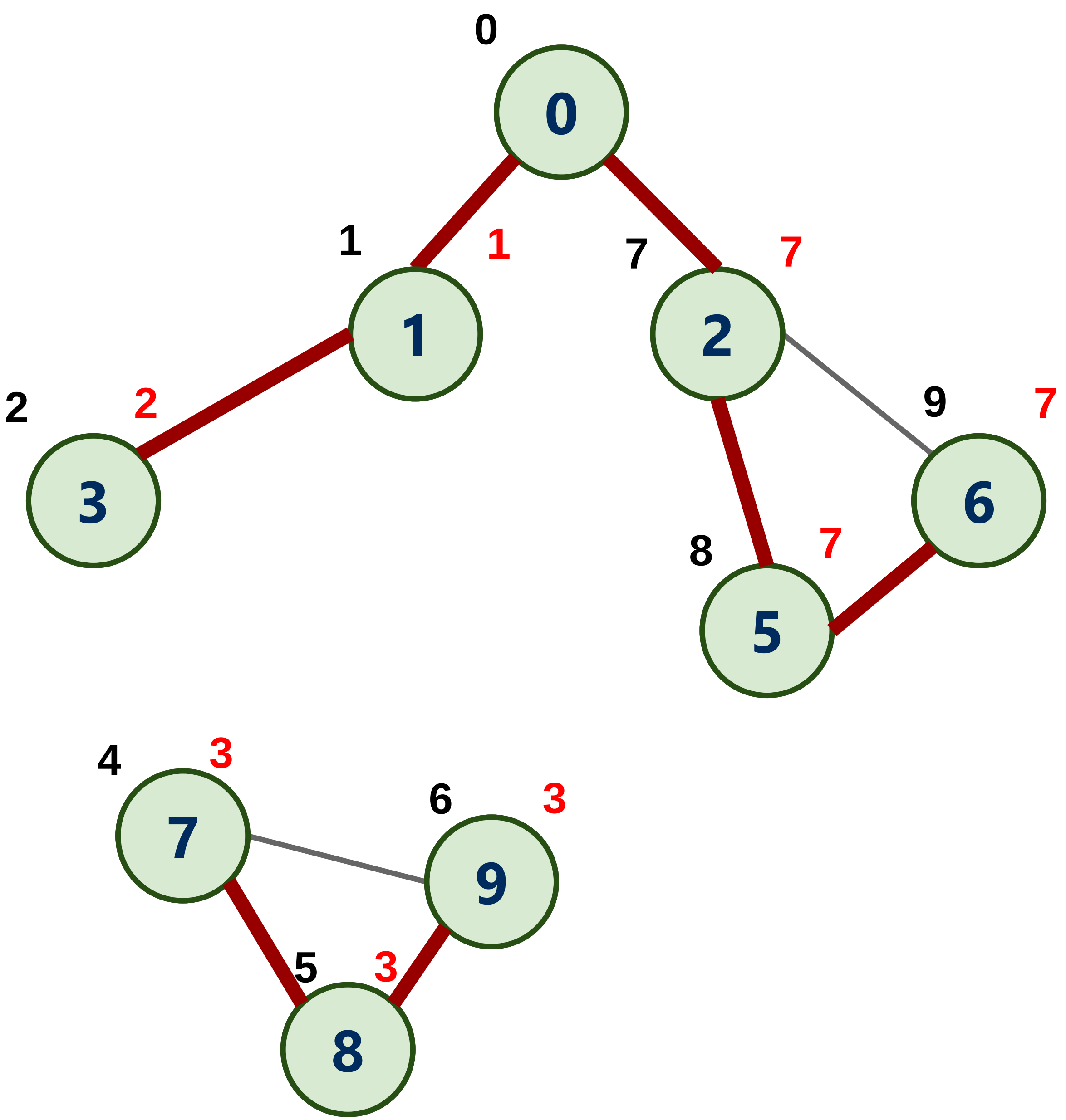
Why are we computing low(v)?

- What does it mean if a vertex has a child such that
  - **low(child) >= num(parent)**?
- e.g., 4 and 7
- child has **no other way** except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the **parent is an articulation point**!
  - e.g., if 4 is removed, the graph becomes disconnected



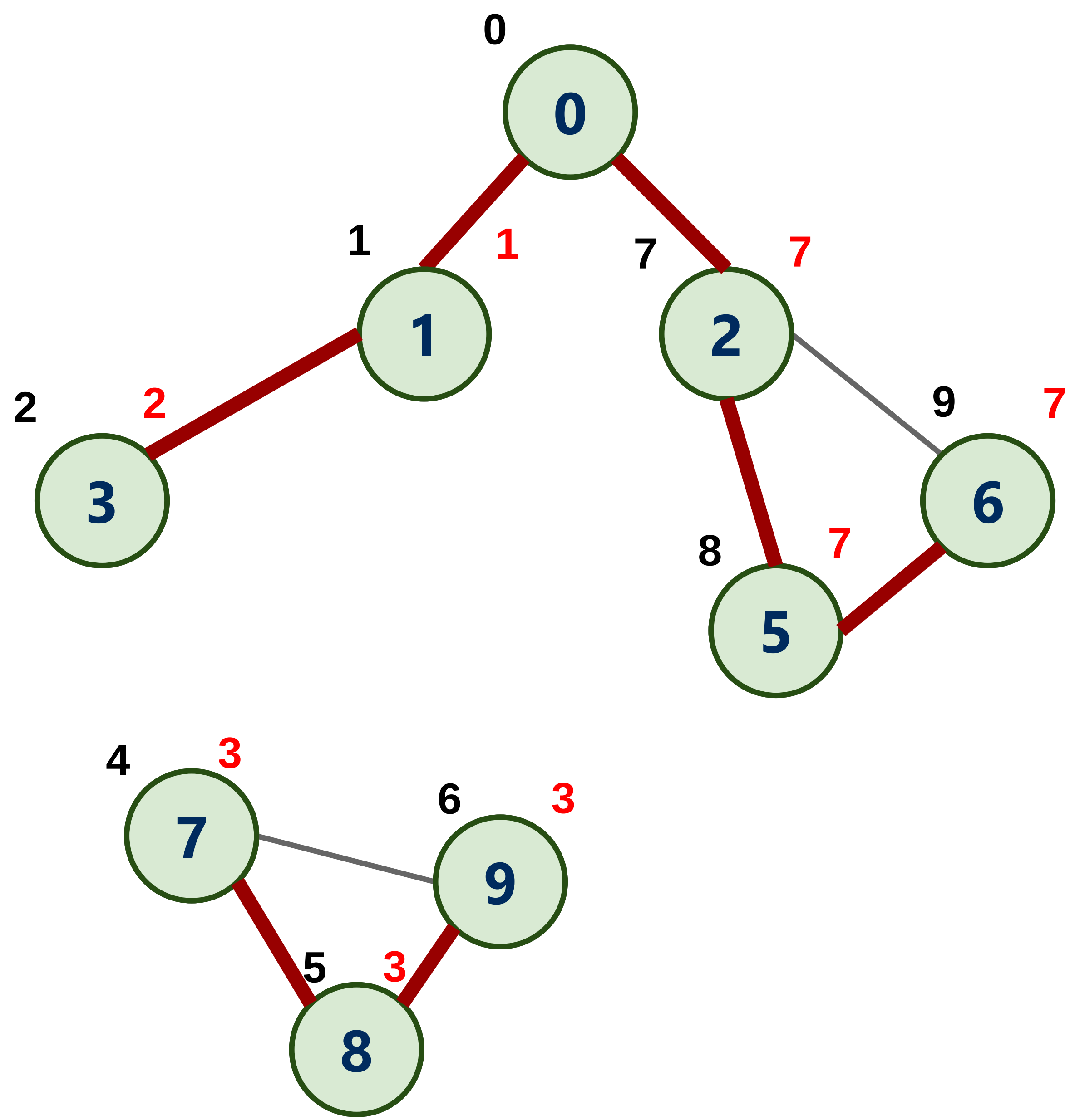
Why are we computing low(v)?

- if 4 is removed, the graph becomes disconnected
- Each **non-root vertex v** that has a child w such that **low(w) >= num(v) is an articulation point**



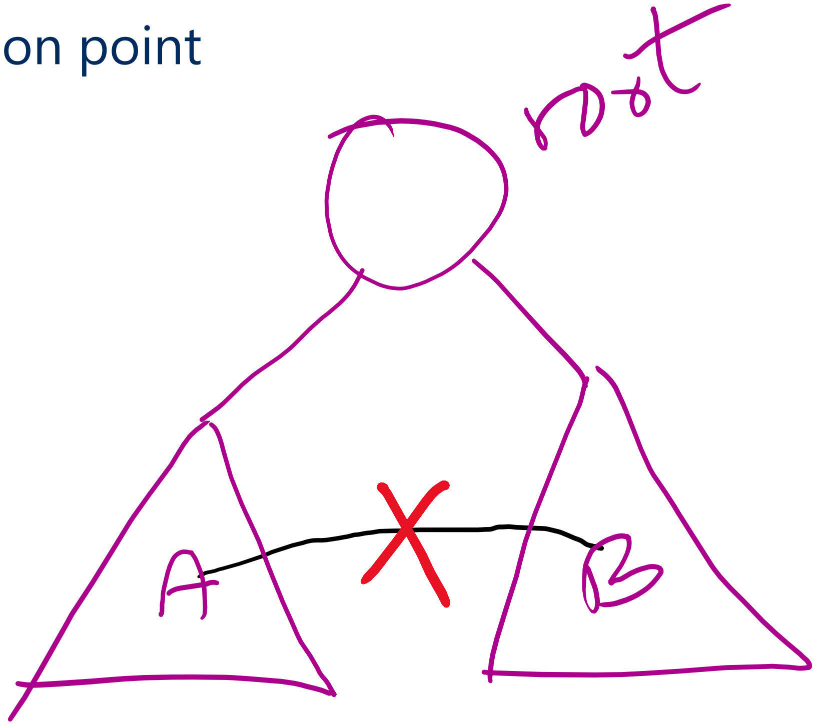
What about the root vertex?

- The root has the smallest num value
  - root's children can't go "further" than root
- Possible that low(child) == num(root) but root is not an articulation point
- need a different condition for root



What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point



Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
  - Label v with with the two numbers:
    - num(v)
    - low(v): initial value is num(v)
  - For each neighbor w
    - if already seen → we have a back edge
      - update low(v) to num(w) if num(w) is less
    - if not seen → we have a child
      - call DFS on the child
      - **after the call returns,**
        - update low(v) to low(w) if low(w) is less

when to compute num(v) and low(v)

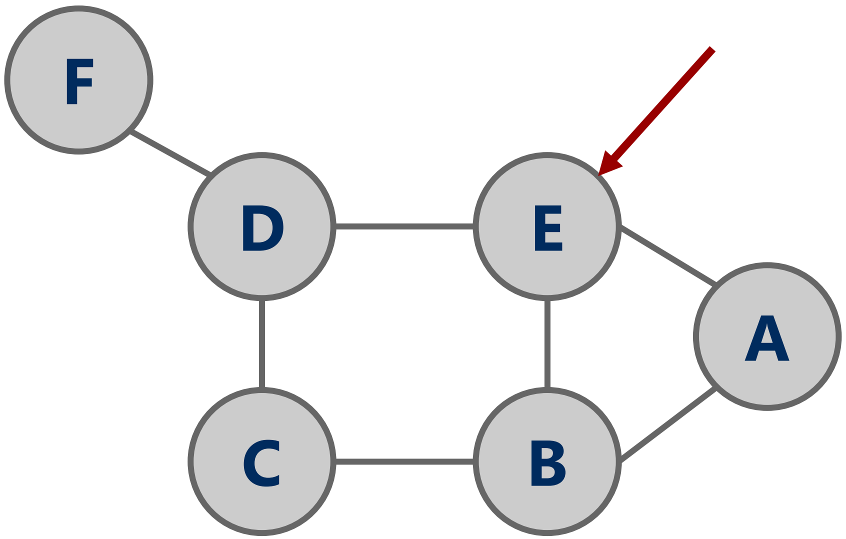
- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
  - why?

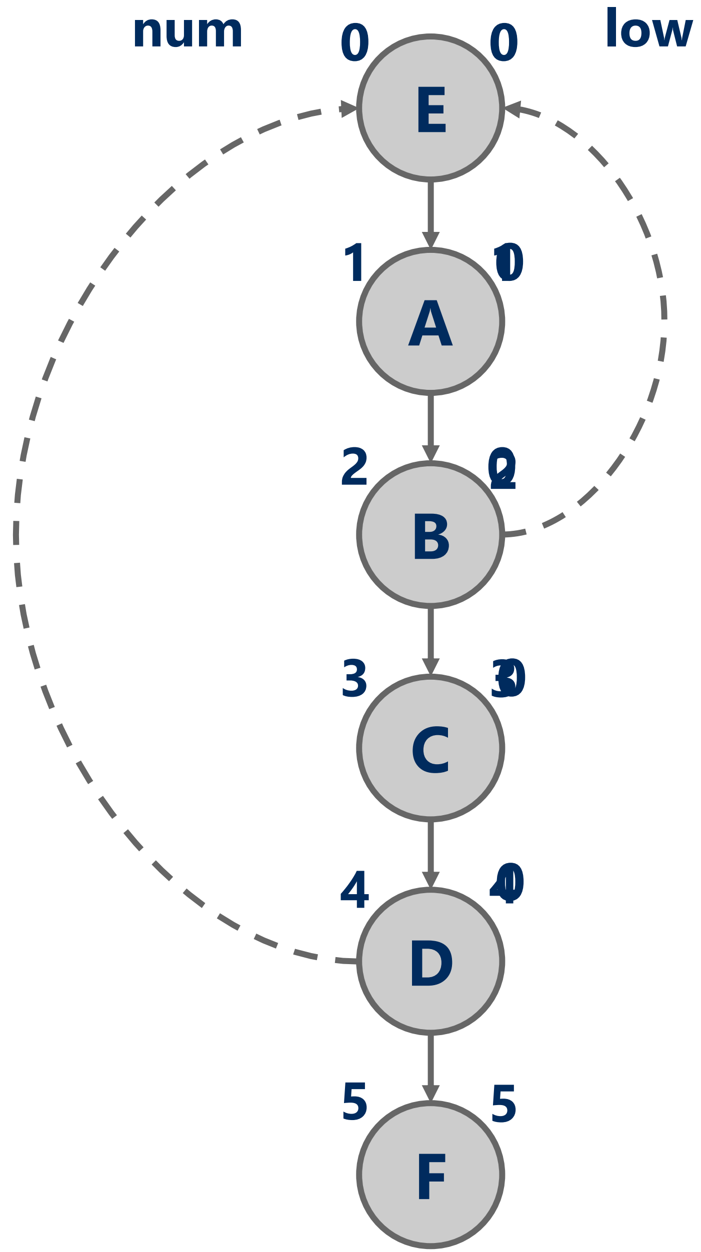
Using DFS to find the articulation points of a connected undirected graph

```
int num = 0

DFS(vertex v) {
      num[v] = num++
      low[v] = num[v] //initially
      seen[v] = true //mark v as seen
      for each neighbor w
         if(w unseen){
            parent[w] = v
            DFS(w) //after the call returns low[w] is computed, why?
            low[v] = min(low[v], low[w])
            if(low[w] >= num[v]) v is an articulation point
         } else { //seen neighbor
            if(w != parent[v]) //and not the parent, so back edge
              low[v] = min(low[v], num[w])
         }
```

Finding articulation points example





Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items
- Output:
  - repeatedly find a minimum item
- You are implementing an algorithm that **repetitively** solve this problem
  - examples of such an algorithm?
    - Selection sort and Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

Let’s create an ADT!

- The Priority Queue ADT

- Let’s generalize min and max to highest **priority**
- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()
- We mentioned priority queues in building Huffman tries
- How do we implement these operations?
  - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - $\Theta(1)$
- Find:
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- Remove:
  - Search for the highest priority item and delete
  - $\Theta(n)$

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- Find:
  - Return the item at the end of the array
  - $\Theta(1)$
- Remove:
  - Return and delete the item at the end of the array
  - $\Theta(1)$

So what other options do we have?

- What about a balanced binary search tree?
    - Insert
        - $\Theta(\lg n)$
    - Find
        - $\Theta(\lg n)$
    - Remove
        - $\Theta(\lg n)$
- OK, all operations are $\Theta(\lg n)$
    - No constant time operations

Which implementation should we choose?

- Depends on the application
- We can compare the *amortized runtime* of each implementation
- Given a set of operations performed by the application:

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$